Code Generation
Part I: The Game Plan

Game Plan

A. Write a few assembly language programs yourself and compile them on the lab machines. Include function calls, loops, and output statements.

B. Write functions that walk through your parse tree and find the offset from fp for every parameter and local variable. You will want to store this information in the declaration nodes of the tree. More about this later

C. Write some helper functions that will format lines of assembly code. For example, here is one of my helper functions:

```
public void genRegReg(String opcode, String r1, String r2, String comment) {
        output.printf( "\t %4s %4s, %4s %10s #%s\n", opcode, r1, r2, "", comment);
}
```

I have about 10 such functions, each formatting different pairs of operands.

Note the comment parameter. You should generate a comment for every line of your output file.

D. Create some string constants for your register names. Among others, I use fp for "%rbx" and sp for "%rsp". You might want to decide now which registers you will use as temporaries.

E. Create a system for generating unique label strings. An easy way to do this is to create a label counter as a static variable and then incorporate that number as part of the label. In Java you can say something like

```
label = String.format(".L%d", labelNumber);
```

F. Write a function that creates the header for your code file. This will ultimately have global variable and array declarations, a .rodata section with string declarations, and a .text section that starts with a .globl main declaration. Start with just the .rodata and .text sections, and in the .rodata section give only the strings you need for printf and scanf. It will look something like the next page:

```
.section .rodata
.WriteIntString: .string "%d "
.WriteInString: .string "\n"
.WriteStringString: .string "%s " .
.ReadIntString: .string "%d"

.text
.globl main
```

G. Your code generator will consist of 3 mutually recursive functions:

genCodeStatement(TreeNode t)

genCodeExpression(TreeNode t)

genCodeFunctionDec(TreeNode t)

You need to build them up simultaneously, in small pieces that allow you to test out the compiler as you go.

H. Start with genCodeFunctionDec(TreeNode t) because the language won't let you write any code that isn't inside a function. This is easy. You need to:
    a) Use the function name as a label for the start of its code.
    b) Move the stack pointer to the frame pointer. I do this with genRegReg("movq", sp, fp, "setup fp");

c) Generate code to allocate the function's temporary variables
d) genCodeStatement(t.body)
e) generate code to deallocate the temporary variables.
f) return

I. genCodeStatement(TreeNode t) needs to look at the kind of statement node t represents. Our function bodies are always compound statements so start there. There is no code to generate for the declaration in a compound statement, so go straight to the body.

J. The body of a compound statement is a statement list. You generate code for this by generating code for each statement in the list.

K. Generate code for a write(exp) node. This calls genCodeExpression( ) on the exp child of the write node. This code should leave the value of exp in the accumulator (%eax or %rax). For the rest of the write statement you should look at the Runtime Environment handout for how to handle I/O.

L. The code generated by
genCodeExpression(TreeNode t)
should always evaluate the expression rooted at node t and leave its value in the accumulator. Start this with integer constants; the code for the expression 23 is just

```
movl $23, %eax
```

This much should allow you to compile and run the program

```
void main(void) {
        write(6);
}
```

M. Now extend genCodeExpression(TreeNode t) to allow arithmetic operations. This allows you compile and run

```
void main(void) {
        write( 3+4*5 );
}
```

N. Everything else will need variables, so extend genCodeExpression(TreeNode t) to handle assignment expressions.

O. I suggest adding function calls and returns next. These are actually easy.

P. The rest of the language you can do in any order, one feature at a time. You might want to save pointers till the end.